"A Tale of Two Provers":

A Comparison of Dependent Haskell and F*

Xinyue Zhang

Advised by Richard A. Eisenberg

March, 2018

Abstract

To be filled

Submitted in Partial fulfillment of the requirement for BA in Computer Science at Bryn Mawr College

Contents

1 Introduction and Motivation

> *"Testing can show the presence of errors, but not their absence."*
> – Edsger Wybe Dijkstra

Programming languages are powerful tools used to design and build complex computer software in order to perform various tasks. Yet complex software programs often have errors commonly known as bugs that prevent them from functioning as expected. Almost everything these days is dependent on software, from smart devices to vehicles, and to national defense systems. Most of the time bugs cause small problems such as an application crash. However, in many cases, tiny errors in a program cause catastrophic if not life threatening consequences such as a financial crisis or a plane crash. Therefore, various research must be done to ensure the correctness of a program.

Through systematic testing and static checking, it's possible to find some incorrect program behaviors, assuming that it is easy or even possible to check the correctness of the output from the program. Yet exhaustive testing like these can only apply to very few trivial programs, and is both labor intensive and program specific, making it not feasible. As Edsger Wybe Dijkstra stated, "testing can show the presence of errors, but not their absence", testing cannot guarantee the correctness of a program. Other options for error checking exists and formal verification is the most widely used technique. Formal verification uses mathematics to ensure the correctness of a software with respect to a defined or inferred specification. We say a software is correct if it meets all of its specifications, which define all its intended behaviors. Formal verification can be applied to programming languages to help verify there are no bugs in the software written. Using formal verification, we want to create a programming language that makes sure all programs written indeed perform as intended. We believe that verification-oriented programming languages have the potential to detect most of the bugs in a program and detect system vulnerabilities at compile time. Therefore, we are looking for better techniques to support program verification in programming languages.

One major approach in formal verification is type system design. Type system is a mechanism that enforces rules on programs through defining type constraints and the associated type applications. A type is an abstract value that classifies a group of values sharing some properties. For example we say that `true` and `false` are both of type `Boolean`. Since the `+` operation is not valid for booleans, we can catch invalid operations such as `true + false` using typing rules. Through designing a formal type system, program specifications can be encoded in the types, allowing the program to become what's known as a "proof-carrying code" [9].

There are three main ways to specify a program, through dependent type, refinement type and Hoare logic (Dependent type and refinement type are discussed in detail in section 2.2 and Hoare logic in section 2.6.4). A dependent type is a type whose definition is predicated upon a dynamic value. As such, dependently-typed programming languages enable programmers to express more detailed specifications with types and to perform more powerful verifications through type checking. On the other hand, refinement type is a type that additionally satisfies a given logical predicates. It helps reduce code redundancy and make specifications more intuitive. For example, consider a length indexed list `l` and let `n` represents the length of the list and `a` represents the type of the list elements. Then `l` is specified through dependent type as `l:list a n`, and through refinement type as `l:list a{len l = n}` with the user-provided `len` function. Finally, Hoare logic is a system of logical rules for reasoning about programs. It consists of Hoare triples that specify the precondition, command of execution, and postcondition such that if the precondition holds before running the command, then the postcondition will hold as long as the command terminates.

Programming languages like Coq [1], Agda [10], and Dafny [7], commonly known as theorem provers, have been studied for a long time and are relatively well understood. They laid down a solid foundation for program verification, but are limited when applied to general-purpose applications in real life. However there are various industrial-strength functional programming languages with resemblance to mathematical functions that also provide a good basis for supporting verifications. Researchers on functional programming languages, like Haskell and Meta Language(ML), gradually start to support verifications with dependent types. Dependent Haskell and F* are two such dependently-typed languages. Each presents a unique and effective program verification technique.

Dependent Haskell and F* are similar in that they both aim towards a verification-based yet general-purpose programming language. They are, however, noticeably different in their unique approaches to verify potential divergent functions. Dependent Haskell doesn't require termination checking, whereas F* categorizes divergent functions into a special `DIV` effect (See section 2.5 and 2.6.4 for details). Most dependently typed verifiers require compile-time termination checking to ensure the correctness of proofs. However, as Alan Turing proved in the Halting Problem, given an arbitrary program and its inputs, checking whether this program will terminate or loop forever is an undecidable problem. Yet functions that diverge or hard to prove termination are very common in practical programming languages. Therefore, we are interested in exploring how to better express and reason about them.

In this undergraduate thesis, we aim to introduce the two verification-oriented programming languages, Dependent Haskell and F*, present their unique type system designs with concrete examples, and give a detailed comparison from a combination of theoretical and practical standpoints. We expect our research to setup a solid foundation for potential collaborations across the two communities, and to possibly propose promising future research directions in formal verifications for the general programming languages community.

2 Background

2.1 From Imperative to functional programming

Most programmers come from a Java or C/C++ background. These languages are designed to primarily support imperative or procedural programming. In the imperative style of programming, a developer writes code that details each step for computers to execute. These imperative programs often have side effects that modify program inputs or global variables. In the programming languages field, any function modifying global states in execution is called stateful. For example, global variables can be modified in any function in scope. Functional programming, as its name suggests, is a style of programming that takes a functional approach. In functional programming, a developer models the problem as a set of functions to be executed, detailing the inputs and outputs for each function. Common functional programming languages are Haskell, ML, Scala, and Lisp. Functions implemented in these languages can be understood as a generalization over high school algebra. Instead of limiting the operations purely on numbers, one abstracts the idea of function and allow it to operate on any structured data types, such as boolean, string, lists, etc. Functional programs are usually executed by evaluating expressions and typically avoid mutating program states. In functional programming, functions are first-class citizens, i.e. they are treated like any other values and can be passed as arguments to other functions or be returned as a result of a function. The functions that take functional arguments are called higher-order functions.

One major difference between the imperative and functional style of programming is on handling program iterations. Imperative programming languages usually support both loops and recursions, but functional programming languages depend highly on recursions and higher-order functions, like `map`, `filter` etc. For example, if one wants to double every element of the list `[1,5,2,4,3]`, one can implement in Java, an imperative programming language as

```
public void double(int[] arr) {
    for(int i = 0; i < arr.length; i++) {
      arr[i] = 2 * arr[i];
    }
}
```

The above code is very intuitive to understand, but notice that the input list `[1,5,2,4,3]` is directly modified after the execution of `double` to the result `[2,10,4,8,6]`.

The same operation can be defined in Haskell in functional style with the higher-order function `map` as,

```
double :: [Int] -> [Int]
double = map (*2)
```

In functional programming style, programmers often declare function signatures on the first line and give actual implementations on the following lines. In the `double` example above, the input is specified as `[Int]`, denoting a list of integers, and output as `[Int]`. The entire type definition `[Int] -> [Int]` after the double colon belongs to the type level. Specifically, when apply `double` to the input list as in `double [1,5,2,4,3]`, the function `(*2)` is mapped to every element of the input list, returning the doubled list `[2,10,4,8,6]` as a result without modifying the input list. Instead of using parenthesis to symbolize function application, one can simply put a space between function name and its parameters. The function, `(*2)`, applied here uses the idea of currying. The expression `(*2)` is equivalent to the familiar lambda notion `\x -> x * 2`, as both functions take a number and double it. The idea of currying is widely used in functionally styled programming, since functions with too few parameters will simply return another function that takes as many parameters as left out. Here, `*` has type `int -> int -> int`, i.e. taking two integer inputs and returns an integer output. As only one parameter is applied to `*` when calling `(*2)`, this expression essentially becomes another function that takes in the remaining integer parameter and returns the product of that integer and `2`. Finally, applying the higher-order function `map`, the doubling operation is mapped on every element of the list, doubling the entire list.

2.2 Program Specification with Dependent and Refinement Types

As introduced in the introduction section, the starting point of any program verification is to specify its properties. Programs can be specified using either dependent types or refinement types. To illustrate these definitions in more detail, consider the example of defining a list. In imperative programming languages like Java or C/C++, elements in a list must all share a single type, say `bool`, and then list is defined with type `bool[]` or `bool*`. If the list is defined on the stack, then a list of length 5 will be declared as `bool[5]`; otherwise, `5 * sizeof(int)` of memory will be malloc'ed on the heap with a pointer pointing to the memory address returned. To specify the list carrying its own length in verification-oriented programming languages, one can predicate the type of this list of booleans v on its length 5, as in the type `Vec 5 Bool`. `v:Vec 5 Bool` is said to be dependently-typed as $v's$ type is indexed on the number 5 of another type `Nat` (the type `Nat` represents the natural numbers, defined to be non-negative integers). Similarly, this list of booleans of length 5 can be specified through logical predicates using refinement types. Recall that refinement types specify a logical formula that the types have to additionally satisfy. So one first defines a function `len` calculating the length of a list and then applies it in the predicate to produce the refinement type `v:vec Bool{len v = 5}`. The logical formula `len v = 5` is called the subtyping predicate of the refinement type. Both dependent types and refinement types are commonly applied in program verifications. Dependent Haskell only supports dependent types, but F* supports both dependent types and refinement types. We will compare them in detail in our result section.

2.3 Programming Effects

Most verification techniques nowadays are based on pure evaluations, receiving inputs and directly returning outputs without modifying any other program states. However, industrial-strength programming in the real world often has side effects. A practical program usually doesn't just return a value output purely from operating on its inputs, but instead might run forever without termination, modify other states beyond its scope, or interact with its calling function or the outside world through streams and sockets. An effect system is a formal system which describes the computational effects of computer programs, such as Inputs and Outputs (I/O), error handling, stateful processing (mutable values in respect to times), and sometimes divergence (non-termination) etc. An effect system can be used to provide a compile-time classification of the possible effects of the program.

2.3.1 Monads

In most functional programming languages, one common way to express programing effects is by using monads. Through monads, programmers can structure computations in terms of values and sequences of computations using these values. Without interacting with the external states, monads allow effectful computations to be embedded in pure languages.

The term monad might be overwhelming for programmers new to Haskell, but it's actually a simple idea wrapping a value into the wrapped value, like preparing a raffle roll. Additionally, monads support unwrapping the wrapped value to reveal the enclosed value. Suppose that one wants to write a function to calculate the square root of an integer and to return the result in double. One attempts is to specify the function with type `sqrt :: Int -> Double`. Yet one quickly find himself in trouble as negative integers don't have real square roots. In languages like Java or C/C++, one can resolve the complexity by simply returning `null` for any negative input. However, functions in Haskell must always return values of the same type, so the untyped value `null` is not supported in strictly typed Haskell. Instead, Haskell introduce the `Maybe` monad, defined as

```
data Maybe a = Just a | Nothing
```

that could either represents a wrapped value incorporating a successfully evaluated value or denoting the state of potential failure.

In the `sqrt` example, successful evaluation of the nonnegative square roots can be denoted as `Just [result]` where `result` is the resulting square root. Whereas the invalid operation on negative integers can return the value `Nothing`, representing the evaluation failure. The `Nothing` value can be seen as the analogue to the special `null` term.

Beyond the `Maybe` monad, Haskell defines the `IO` monad for handling inputs and outputs, `Error` monad for exception handling, `State` monad to simulate in-place value updates etc.

2.4 Dependent Haskell

2.4.1 Learn You a Haskell

Haskell is a statically-typed and strongly-typed purely functional programming language. It features a type system with type inference and lazy evaluation.

As a pure functional programming language, each function in Haskell only takes in some inputs and returns an output after some operations, without modifying any unspecified program states. Unlike many functions (which should really be called procedures) in an impure language, functions in Haskell could only observe the inputs defined. That is to say that in the execution of every Haskell function, no variables will be mutated; no reading and writing to standard inputs and outputs or connecting to a socket is allowed; and no error handling is performed etc. This seemingly limiting feature ensures that each execution of the function always returns the exactly same result, which allows formal proofs on functions and also composing functions to perform more complex operations.

Haskell is statically typed, where every variable's type is determined at compile time. In addition, it has a type system that ensures every variable type to be checked before performing any operations. It also enforces strict typing rules on functions, so called strongly-typed. In strongly-typed languages, every variable type is predefined and requires explicit casts when used differently. By this definition, Java is strongly typed since it rejects operations like `1 + "Haskell"` but C is weakly typed since it accepts the mixed-typed operations like `1 + false`, treating `false` as `0`.

Even though every single type is strictly enforced, Haskell still remains elegant and concise. It uses a type system with full support of type inference, enabling programmers to leave out many type definitions. For example, code pieces like `a = x + 2` that adds an integer variable `x` to a numeric literal `2` is allowed without explicitly specifying 2's type. Haskell can deduce 2 to have type `Int` from the operation `(+)`, since `(+)` supports only addition of numbers of the same type.

Finally, Haskell is lazy, since every Haskell operation is lazily evaluated. This means that the final result of an operation is only computed unless they're required. Suppose we want to quadruple the list `l = [1,2,3,4]` using our afore-defined `double` function by calling `r = double(double [1,2,3,4])`. In non-lazy languages, the two `double` expressions will be evaluated right away to produce the resulting list `[4,8,12,16]` and assign it to `r`. However, in lazily evaluated languages, evaluations are deferred until results are needed by other computations. For example, the evaluation will occur when the programmer wants to get the first element in the resulting quadrupled list. Specifically, the first `double` will request a result from the second inner `double` and the second inner `double` will be executed to produce the doubled list, and finally the outer `double` takes the resulting doubled list as input and produce the quadrupled list and returns its first element. The biggest advantage of a lazy language is to improve code modularity without sacrificing code efficiency. In the programming languages field, laziness is often referred to as the call-by-name evaluation scheme.

2.4.2 Dependently-Typed Dependent Haskell

For a long time, Haskell researchers have strived to add dependent types in Haskell, as dependent types introduce much more precise expressions of program specifications. Although Haskell doesn't yet support dependent types in full, several extensions to its current type system have made exceptional progress.

In 2012, Eisenberg and Weirich introduced the `singletons` library that simulates dependently-typed programming in Haskell. Through dependent types, programmers can reason about programs through computations on the indexed types. For example, when concatenating two lists of booleans, each of length `m` and `n`, the output list can be specified to have type `vec (Plus m n) Bool`, indexed on the type variable `Plus m n` representing the sum of the two lengths of the original lists. However, Haskell enforces two separate scopes in programming functions – the terms executed at run time and the types checked at compile time. Functions in these two scopes cannot be used interchangeably, so singleton types are necessary to break the gap. Specifically, with the Singletons library, the `Plus` function defined in the term level can be automatically promoted to the type level and applied to reasoning about the dependent types. In addition, Singletons are defined to take in a general type and promote it to reason in the type level, as `SNat n` brings the index `n` of type `Nat` to type `Type` that can be used to retrieve the `n`th element from a list.

Beyond Singletons, Gundry and Eisenberg both conducted research on dependently-typed Haskell using GADTs and type families (Detailed definitions and examples are provided in section 2.4 Program Verification). Eisenberg, extending Gundry's work, brings Haskell to a full-spectrum dependently typed language. Eisenberg's work, now implemented in the Glasgow Haskell Compiler (GHC) 8.0, finally blurs the distinction between type and kind, unifying expressions and types in Dependent Haskell. Unlike Gundry, who limits type promotion to exclude lambda expressions and unsaturated functions, Eisenberg enables promoting any expression available at run time to type [3]. To clarify the relationship between type and kind, consider the number `5`, which is a term. Similar to any strongly typed languages such as Java, `5` has type `Int`. Then, one can go one level up and classify all types, including `Int`, `Bool`, `Vec 5 Bool` etc, to have type `Type`. Every data type then has type `Type` (or denoted as `*`), which is analogous to every integer values of type `Int`. In short, kinds can be understood as the type of types, which classify types, as how types classify terms.

Besides the unification of expressions and types, Eisenberg introduces the `'` operator to bring ordinary term-level functions directly into type-level, and the universal quantifier $\forall$ to enrich the logic. Another major contribution of Eisenberg's work is pattern matching on dependent types, such as pattern matching on the indexed type `n` in the dependent type `Vec n a` (where `n` is the length of the vector and `a` is the type of its element).

Finally, Dependent Haskell continues as Haskell to be a partial language, without compile-time termination checking. Dependent Haskell only guarantees correctness of fully evaluated proofs, thus only allowing the well-typed values. For example, though erroneous proofs like `1 :~: 2`, a propositional equality GADT representing the logic $1 = 2$, might be verified, they will never be fully evaluated down to values. The erroneous logic `1 :~: 2` won't be evaluated to value since it doesn't follow the rules defined in Dependent Haskell's type system, therefore not well-typed.

2.5 Program Verification in Dependent Haskell

2.5.1 Soundness and Completeness in Type Systems

In type system based program verification, one often wants to reason about two main properties of the language's type system, as soundness and completeness. A system is sound if every property that can be proved in the system is logically valid with respect to the semantics of the system. On the other hand, A system is complete if every logically valid property with respect to the semantics of the system can be proved by the system. Generally, programming languages researchers strive to make a type system more complete without sacrificing the soundness of each proof.

2.5.2 Algebraic Data Types

Type system based program verification is all about types. Beyond the primitive types, `Int`, `Bool` etc, defined in the standard library, one can also define our own types. Similar to the C structs, new types can be introduced through the `data` keyword.

For example, a list of elements of type `a` is defined recursively in Haskell as:

```
data List a = Nil | Cons a (List a)
```

The library list type is defined above with two value constructors, `Nil` and `Cons`. Each constructor specifies a value that the list type could have. Specifically, `Nil` is the base case representing the empty list and `Cons`, often written as `:>`, is the inductive step that combines one more element to an existing list. The symbol `|` represents logical or, so the above code means that the `List` type whose elements are all of type `a` can have values either an empty list or a combination of an element of type `a` and another list of elements of type `a`. In functional programming and type theory, any type defined with value constructors is called an algebraic data type(ADT).

Similarly, one can formally define the natural numbers recursively in Dependent Haskell as:

```
data Nat where
  Zero :: Nat
  Succ :: Nat -> Nat
```

where `Zero` represents the smallest natural number 0, and `Succ` represents the nonzero numbers as they are all increments of a smallest natural number.

2.5.3 Generalized Algebraic Data Types

It follows that, using dependent types, one can encode the length of a list into its type as a natural number, by parameterizing the list by its length and its element type. Below we introduce the dependently typed vector definition in Dependent Haskell:

```
data Vec :: Nat -> Type -> Type where
  Nil  :: Vec Zero a
  (:>) :: a -> Vec n a -> Vec (Succ n) a
infixr 5 :>
```

Instead of a simple listing of value constructors, the definition of `Vec` extends that of `List` by explicitly clarifying the type signature for each constructor, instantiating the algebraic data type (the dependently typed `Vec Zero a` and `Vec (Succ n) a`) as return values. Notice that both `List` and `Vec` introduced resemble closer to the linked list data structure. Since functional languages don't modify any state of the input list, input list is often copied over to form the output list. In the above definition, the `:>` symbol represents the Cons operator as before that combines an element with another vector to produce a new vector. Also, `infixr 5 :>` defines the `Cons` operator to be right-associative, i.e. `1 :> 2 :> 3` will be evaluated as `1 :> (2 :> 3)` and results in the vector `[1, 2, 3]`. In general, an ADT whose value constructors have explicit type instantiations and match promoted term-level ADT patterns to refine type-level statements is called a Generalized Algebraic Data Type, or a GADT.

2.5.4 Type Class

Finally, we introduce the concept of a typeclass. Typeclasses can be understood as an interface that categorize a group of behaviors. It contains many types that all support and implement the behavior the associated typeclass describes. For example, `Ord` is a typeclass for all types that have an ordering, for example `Int`, `Double` etc. All elements of the type that have an ordering can be compared to check if one is greater than, equal, or less than another. To specify the usage of a certain type class, take `Ord` as an example, programmers can pass in the type variable `a` as in `Ord a` to specify that type `a` is part of the `Ord` type class and has an ordering.

2.5.5 A Simple Verification Example

With the introduced concepts as parametrized dependently-typed GADT `Vec` and the `Ord` typeclass of all types with orderings, we begin with a simple example on how program verification works in Haskell.

Specifically, one can define the `insert` function in Dependent Haskell to insert an element to a vector.

```
insert :: Ord a => a -> Vec n a -> Vec (Succ n) a
insert elm Nil       = elm :> Nil
insert elm (x :> xs) = if elm > x then x :> (insert elm xs) else elm :> x :> xs
```

Elements in the list of type `a` are all comparable. Since in the type signature `insert :: Ord a => a -> Vec n a -> Vec (Succ n) a`, we define `a` to belong in the `Ord` type class, as in `Ord a =>`. Also, the above implementation uses GADT pattern matching to classify which case to apply. Specifically, `Nil` is to pattern match the original input list with the empty list, and `(x :> xs)` is to pattern match the original input list with at least an element. The above insert algorithm may be understood imperatively as follows: the element `elm` is inserted to the list if the list input is originally empty. Otherwise, as the list contains ordered elements, one may compare the value of `elm` with the smallest element `x` in the original nonempty list, and either recursively insert the new element into the original list or directly prepend the element to the front of the list. Notice that when the above `insert` algorithm is repeatedly applied, the eventual output list will be sorted.

Similar to writing test cases, program verifications also focus on the specification of each program. For insert, the length of the list is incremented by 1 every time one inserts a new element. Therefore, the return type for function `insert` can be defined as `Vec (Succ n) a`, encoding the length incrementation in the returned GADT.

2.5.6 Type Family

To perform some more involved computations in type level, such as addition or multiplication, Dependent Haskell introduce what's known as a type family. For example when concatenating two `Vec`s, one needs to verify the specification such that the resulting `Vec` has length the sum of the two concatenated `Vec`s.

Type family is a function defined at type level that returns a type. It gives us an extended ability to compute over types and is the core of type-level computation. For example, to add two type-level `Nat`s together, we define the type family in Dependent Haskell as:

```
type family Plus (a :: Nat) (b :: Nat) :: Nat where
  Plus Zero b      = b
  Plus (Succ a') b = Succ (Plus a' b)
infixl 6 Plus
```

The implementation of `Plus` is exactly the same as the ordinary `(+)` implementation in the term level, except that `Plus` is one level up in the type level. Also, `infixl 6 +` defines the `(+)` operator to be left-associative, i.e. `1 + 2 + 3` will be evaluated as `(1 + 2) + 3` on default. Again, we apply pattern matching on the first summand of plus. If the first summand `a` is `0`, then we define our base case such that any sum of `0` and another natural number `b` is just the other natural number `b`, or in short `0 + b = b` for all natural number `b`. Otherwise, if the first summand is a non-zero natural number, represented as `Succ a` where `a'` is the natural number one less than `a`, then one recursively applies `Plus` on the smaller natural number `a'` and the other natural number `b` to obtain the sum and add one to obtain the result. Equivalently, in algebraic equations, if `a = 1 + a'` then the operation is essentially `a + b = (1 + a') + b = 1 + (a' + b)` from associativity of plus.

2.5.7 Proof Verification with Dependent Types and Type Families

Finally, applying both dependent types and type families, one can verify the operation of concatenating two lists. The operation `concat` can be recursively implemented and verified in Dependent Haskell as:

```
concat :: Vec n a -> Vec m a -> Vec (Plus n m) a
concat Nil v2       = v2
concat (x :> xs) v2 = x :> (concat xs v2)
```

The implementation is similar as before, applying pattern matching on the first parameter and recursion on the smaller list. After concatenating two lists, the resulting vector is expected to have length the sum of the lengths of the two original vectors, as specified in the dependent types `Vec (Plus n m) a` applying the type family `Plus` in the typing level.

2.6 FStar

F* is an ML-like functional programming language aimed at program verification [13]. It is developed at Microsoft Research, MSR-Inria, and Inria. F*'s syntax is similar to OCaml and F# in the ML family. ML is short for the Meta Language invented for proving theorems. In 2016, Swamy et al. introduced the current, completely redesigned version of F*. The new F* is a general-purpose, verification-oriented and effectful programming language. F* is said to be general-purpose as it is expressive enough to support programming in various practical domains; it is verification-oriented as it applies formal verification techniques to prove the correctness of programs written in F*; F* is also effectful, for it allows programs with side-effects, such as I/O, exceptions, stateful processing, or programs that potentially diverge.

2.6.1 Introduction to F*

2.6.1.1 F* Basics

As a functional programming language, F* shares most concepts with Haskell, like statically typing, strictly typing, currying etc. There are only small differences between F*'s syntax and that of Dependent Haskell. Below is an example of F*'s implementation of `factorial`:

```
val factorial: x:int{x>=0} -> Tot int
let rec factorial n =
  if n = 0 then 1 else n * (factorial (n - 1))
```

In F*, programmers also specify the function signature before the actual implementation. All function signatures start with the keyword `val` and the type signature is introduced after a single colon instead of a double colon in Haskell. Also, all function implementations start with the keyword `let` and if the function is recursively called, the keyword `rec` will be specified between `let` and the function name.

F* support both dependent types and refinement types. Instead of defining a new GADT for natural numbers, in order to restrict the input of `factorial` to natural numbers, one can also refine the type of `x` to `int{x >= 0}`. This type with logical predicate `x >= 0` is called a refinement type. Also notice that the function `factorial` doesn't just return an `int` but `Tot int`. The prefix `Tot` is a special derived form of the `PURE` effect, categorizing all pure functions(the concept of pure is defined in section 2.3.1). This `Tot` effect specifies the function `factorial` to be total, i.e. guaranteed to terminate. As an effectful language, F* categorizes the programming effects and in addition divergence in a lattice, where some effects are part of other effects. For example, the effect `PURE` on total functions is a sub-effect of the divergent effect `DIV`, which is again a sub-effect of the exception effect `EXN` and finally the effect `ALL` categorizes every effect defined [13]. Specifically, Swamy et al. define these effects as monads that are used by F* to discharge verifications using both the Satisfiability Modulo Theories (SMT) solver (discussed in section 2.6.3) and manual proofs.

2.6.1.2 Simple Inductive Types

Similar to Dependent Haskell, F* allows creating new types using the `type` keyword. We show the definition of the standard library function `list` as follows:

```
type list 'a =
  | Nil  : list 'a
  | Cons : hd:'a -> tl:list 'a -> list 'a
```

The definition of `list` in F* follows a similar syntax as Haskell. One can use `'a` to denote an arbitrary type(like generics in Java) and use `=` instead of `where` to start the actual type definition. Notice that the type constructors `Nil` and `Cons` both have to begin with a capital letter. The pattern matching is indicated by the `|` keyword and the constructor types follow by a single colon. Finally, F* requires all return types to annotate effects. In both type constructors for `list`, when effects aren't specified, the default effect `Tot` is automatically inferred by F* for the result `list 'a`.

With the `list` definition, the `length` function can be easily programmed to return the length of any input list. Specifically, one applies pattern matching on the input list `l` with syntax `match l with` and uses the `|` symbol to separate cases. The `length` function defined on `list` is widely used in the refinement logic to specify conditions for proving lemmas.

```
val length: list 'a -> Tot nat
let rec length l = match l with
  | [] -> 0
  | _ :: tl -> 1 + length tl
```

There are two new syntax introduced in the `length` definition, two syntactic sugar for the list constructors where `[]` represents `Nil` and `_ :: tl` represents `Cons _ tl`. In the first pattern matching case, one defines the length of an empty list, `Nil`, to be `0`. In the second pattern matching case, one recursively defines the length of a list to be one more than the length of the sublist starting from the second element. The `_` symbol is commonly known as a wildcard, representing the value we don't care about when performing pattern matching.

Finally, the same `concat` function introduced in section 2.4.7 can be defined in F* as follows:

```
val concat : #a:Type -> list a -> list a -> Tot (list a)
let rec concat #a l1 l2 = match l1 with
  | [] -> l2
  | hd :: tl -> hd :: concat #a tl l2
```

In the above implementation, an extra argument `#a:Type` is introduced in the type definition. When preceded by the `#` symbol, the type `a` is turned into an implicit argument. Implicit arguments, always preceded by a `#`, tell the F* type system to try inferring the type automatically instead of always requiring programmers to provide manually. It is a kind of type inference that ease the complexity of writing code in a strictly typed environment. Similarly to the `'a` argument in the `list` definition, the implicit argument `#a` of type `Type` denotes a generic type.

2.6.2 Intrinsic and Extrinsic Style of Verification

F* in general supports two approach for verifying programs. Programmers can prove properties either by enriching the types of a function or by writing a separate lemma. These two ways towards verification are often called the 'intrinsic' and 'extrinsic' style.

2.6.2.1 Proving List Concatenation Intrinsically through Type Enrichment

In the `concat` example for concatenating two lists, one wants to prove the property that the length of the resulting list is equal to the sum of the lengths of the two input list. F* supports verifying the property by indexing on the length of the list using dependent types. With the definition of the `vec` GADT, an implementation of the proof is shown below, similar to the one presented in section 2.4.7 in Dependent Haskell.

```
type vec (a:Type) : nat -> Type =
   | Nil  :  vec a 0
   | Cons : hd:a -> n:nat -> tl:vec a n -> vec a (n + 1)
```

This definition of the dependently typed `vec` is analogous to the `Vec` GADT in Dependent Haskell as defined in section 2.4.3, except that we apply the type parameter `(a:Type)` before colon to bring the generic type `a` in scope for all type definitions below.

Then the verified `concat` function is exactly the same as in Dependent Haskell except the need to introduce two implicit arguments to bring the two list lengths in scope.

```
val concat : #a:Type -> #n:nat -> #m:nat -> l1:vector a n -> l2:vector a m ->
    Tot (vector a (n + m))
let rec concat #a #n #m l1 l2 = match l1 with
  | Nil        -> l2
  | Cons hd tl -> Cons hd (concat #a tl l2)
```

2.6.2.2 Proving List Concatenation Extrinsically through Lemmas

In addition to the intrinsic verification style, F* also supports extrinsic verifications through lemmas. Going back to the weak typing using `list` and with the predefined `length` function, one can prove the lemma as follow:

```
val concat_pf: l1:list 'a -> l2:list 'a
          -> Lemma (requires True)
                   (ensures (length (concat l1 l2) = length l1 + length l2)))
let rec concat_pf l1 l2 = match l1 with
    | []     -> ()
    | hd::tl -> concat_pf tl l2
```

As `concat` is a total function that guarantees to terminate, it can be directly lifted wholesale to the logic level and be reasoned about in the postcondition following the keyword `ensures`. In the `concat_pf` lemma, the property is valid in all conditions, so the precondition following the keyword `requires` is simply defined as `True`. With both precondition and postcondition specified, one can prove the lemma recursively. Notice that the `()` symbol introduce here is equivalent to `Refl` in Dependent Haskell, short for reflexivity, and denotes the satisfaction of the property.

2.6.3 Proof Automation

Many researches have been done to reduce the tedious work through automating the proof processes. Automated theorem proving is a subfield of mathematical logic that proves mathematical theorems through computer programs. Satisfiability Modulo Theories (SMT) solvers provide a good foundation for highly automated programs [8].

Satisfiability Modulo Theories (SMT) problems are a set of decision problems that generalize boolean satisfiability (SAT) with arithmetic, fixed-sized bit-vectors, arrays, and other related first-order theories. Z3 is an efficient SMT Solver introduced by Microsoft Research that is now widely adopted in program verifications [2]. Dependent Haskell doesn't support proof automation at any level, but the F* language achieves semi-automation through a combination of SMT solving using Z3 and user-provided lemmas.

With refinement types, F* can discharge the proof on the `concat` example above automatically using Z3.

```
val concat : l1:list 'a -> l2:list 'a -> Tot (l:list 'a{length l = length l1 +
    length l2})
let rec concat l1 l2 = match l1 with
  | []       -> l2
  | hd :: tl -> hd :: concat tl l2
```

The return type for the list `l` can be specified using the refinement type as `list 'a{length l = length l1 + length l2}`. Then F* will generate verification conditions (VC) based on the refinement type and eventually pass to the Z3 SMT-solver for the verification.

2.6.4 Categorizing effects in F*

2.6.4.1 Dijkstra Monad and Hoare Logic

F* categorizes effects using what's known as a Dijkstra monad. Dijkstra monad is a special kind of monad that incorporates Hoare logic to classify effects into preconditions and postconditions. Hoare logic also introduces what's known as a weakest precondition. The term "weak" here refers to how general an assertion is, so weakest precondition is the most general precondition, i.e. logical predicate on the program inputs, needs to establish the postcondition. For a terminating division function on natural numbers, any predicate excluding zero in the divisor x is a valid precondition, such as `int{x > 5}` or `int{x > 10}`. Yet the weakest precondition can be uniquely defined as `int{x > 0}`.

Swamy et al. observes that in the context of dependently typed programming, the weakest precondition function can be modeled as a monad [13]. We won't go into details of the Dijkstra monads but the core idea is to categorize the weakest preconditions for each effect introduced before, such as `PURE` or `DIV`, as monads. Besides the complicated theories behind the categorization of programming effects, F* provides the `require-ensure` syntactic sugar to help with actual programming. F* programmers only need to provide the function postconditions following the `requires` keyword and preconditions following the `ensures` keyword, and F* can de-sugar the syntax to discharge actual proofs.

Through the effect system designed using Dijkstra monads, F* provides a powerful mechanism, called monadic returns, to enrich function types extrinsically. Specifically, for any total function `e` that can be proved to terminate, both its type and definition can be lifted into the logic and reason about it either in the subtyping predicates of refinement types or in the postconditions following keyword `ensures`.

2.6.4.2 Proving Termination

As we are mostly interested in how F* handles potentially divergent functions, we will focus on the `Tot` and `DIV` effect and introduce F*'s approach towards termination checking.

F* by default requires termination checking for every function, since divergent functions may generate unsound verification conditions (VCs) and eventually end up proving some erroneous logic. F* introduces a new termination criterion based on a well-founded partial order $\prec$ over all terms. A partial order is a binary relation on a set that is reflexive, antisymmetric and transitive. For example, the `<=` ordering on integers is a partial order, since for every integer `a`, `b` and `c`, it follows that `a <= a` (reflexive), `a <= b` but `b >= a` (antisymmetric), and finally if `a <= b` and `b <= c` then `a <= c` (transitive). F* insists each iteration of a recursive function be applied on a strictly smaller domain, such that the parameter of the function is guaranteed to decrease in the defined partial order during each execution. The F* type-checker provides four built-in well-founded ordering that can be automatically applied in proving termination, but it also accepts decreasing metrics explicitly provided by the programmer following the `decreases` keyword. Below we summarize these four default metrics and provide an example in each:

(1) The natural number ordering: For instance, the natural number parameter in the `factorial` example in section 2.6.1.1 is decreasing in every iteration.

(2) For any function `f: x:t -> Tot t'`, the curried function (See section 2.1 for the discussion on currying) `f v` for some `v:t` is "less than" the fully defined function `f`. For instance, consider the `length` function introduced in section 2.6.1.2, we would have `length (1 :: 2)` $\prec$ `length`. This metric on currying is very rarely used, so we won't go into details. Instead of applying this metric on currying, the termination proof on `length` is through default metric (3) discussed next.

(3) The sub-terms of an inductively defined term is less than the term itself: For instance, in the `concat` example in section 2.6.2.1 and 2.6.3, every recursively call on the tail of the input list (sublist starting from the second element) is smaller than the original call on input list since the length of the tail is shorter than that of the input list.

(4) The lexicographical ordering: By default, F* checks the decreasing metric on each non-function-typed argument in its order of application. For instance, in the `concat` example in section 2.6.2.1 and 2.6.3, if the order of `l1` and `l2` is switched but the function definition still pattern matches on `l1`, then F* will first check the decreasing metric on the first parameter `l2`. Since `l2` stays the same, the termination checker continues to check the second parameter `l1` whose length indeed decreases by default metric (3). Hence the

`concat` function with parameters switched is still proved total.

When the function is proved total, one can specify it using the `Tot` effect, notifying the F* effect system that the function is guaranteed terminating. If, instead, one can't find any termination metric or the function indeed diverges, F* provides the `DIV` effect to categorize all these potentially divergent functions. Once a function is marked as `DIV`, F* will turn off all termination checking on that function.

3 Approach and Novelty

To gain a better understanding of the language design decisions and verification techniques, We first carefully study the literature of each language. In general, our assessments of Dependent Haskell and F* is based on implementing and proving a wide spectrum of algorithms in both languages. These programs are written so that we could have a much more in-depth experience with both languages and explore their features more thoroughly. To begin with, in both Dependent Haskell and F*, we implement the data type of a length-indexed vector and verify various functions that support types of finite length vectors from Haskell's DataList module, including `length`, `head`, `tail`, `init`, `last`, `snoc`, `reverse`, `and`, `or`, `null` etc. We then focus on the verification for various sorting algorithms, such as insertion sort, mergesort and quicksort, and examine the capability of proving divergent functions, as in the proof of the Peano division property `(a * b) / b = a` using the divergent definition of division with zero divisors and the proof of the equivalence of multi-step and big-step evaluators in simply typed lambda calculus (Discussed in section 4 Current Results).

We expect to structure the comparison mainly focusing on two aspects: user interface and approaches towards potentially non-terminating functions. As far as we know, we are the first researchers to directly compare the two languages, Dependent Haskell and F*.

User interface refers to the interaction between programmer and the programs, specifically programming simplicity when implementing programs in Dependent Haskell or F*. We are interested in examining how these two language designs differ in respect to the ease of use measured from syntactic verbosity and library/documentation resources. As both languages are verification oriented, in addition to general programming syntax, we would also include a comparison on proof complexities for each verification technique. We believe that a good programming language should provide a clear, concise and user-friendly interface with relatively detailed library and documentation supports.

Approaches towards potentially non-terminating functions is measured from the soundness and completeness of the type systems. As introduced in Section 2.4.1, every property proved in a sound system is guaranteed valid in logic and every logically valid property with respect to the system's semantics can be proved in the system. Both type systems in Dependent Haskell and F* is sound and complete when we restrict the range of functions to those that are provably terminating, called total functions. However, when we extend the system to consider potentially non-terminating functions, either functions that diverge or are provably hard to prove convergence, ensuring both soundness and completeness in the type system is a huge challenge. Without compile time termination checking, Dependent Haskell seems to be complete and is able to prove properties on logically valid properties for divergent functions, yet it's not sound as it verifies some logically erroneous proofs (Research in progress). On the other hand, F* is provably sound by ensuring termination checking on all total function of the `Tot` effect, yet it's unknown whether F*'s type system is complete with its categorization of potential divergent functions using the `DIV` effect.

4 Current Results

4.1 Efforts To Date

As summarized in Section 3, we have implemented various functions that support types of finite length vectors from Haskell's DataList module, three machine-checked sorting algorithms – Insertion Sort, MergeSort and QuickSort, and a proof on Peano division with a divergent division definition.

4.1.1 User Interface: General Language Design and Support

4.1.1.1 Syntactic Verbosity

In terms of general language design and support, we conclude that Dependent Haskell is syntactically verbose while F* has a fairly friendly user interface.

In Dependent Haskell, due to the phase separation in term and type level, programmers are often expected to implement the same logic twice. The first type of duplication is in data type definition. Dependent Haskell requires the use of `singleton`, which captures the same logic of a GADT, but promoted to the type level. As such, the exactly same definition for `Peano` data type must be reimplemented using `singleton` as in `SPeano`:

```
-- term-level Peano definition
data Peano n where
Zero :: Peano Zero
Succ :: Peano n -> Peano (Succ n)

-- type-level singleton SPeano definition
data SPeano n where
SZero :: SPeano Zero
SSucc :: SPeano n -> SPeano (Succ n)
```

The second type of duplication comes from function reimplementation using type families. Functions such as adding two natural numbers, need to be specified once in the term level, but exactly the same implementation again in type level using type family:

```
-- term-level plus definition with the singleton SPeano type
plus :: SPeano m -> SPeano n -> SPeano (m + n)
plus SZero n     = n
plus (SSucc m) n = SSucc (plus m n)
infix 6 plus

-- type-level Plus definition using type family
type family Plus (a :: Peano) (b :: Peano) :: Peano where
  Plus Zero b     = b
  Plus (Succ a) b = Succ (Plus a b)
infix 6 Plus
```

On the other hand, F* is syntactically concise. It supports both type-level specifications through dependent types and type-level computations through refinement types. For example, the type signature for the `concat` function introduced in section 2.4.7 can be specified using dependent type as `val concat: #a:Type -> #n:nat -> #m:nat -> Vec n a -> Vec m a -> Vec (n + m) a`. F*'s approach towards dependent types is similar to that of Dependent Haskell, except that F* frees programmers from the repetitive type family implementation of `Plus`. Through the `T-Ret` typing rule using the idea from monadic returns (we won't go into detail for the typing rule, but a brief introduction is provided in section 2.6.4.1), F* can automatically enrich the types of a total function and reflect the existing term-level implementations into logic for reasoning. Monadic returns in F* are similar to the `return` typing rule in monads, returning the weakest precondition categorized as a monad after proving the post-condition on it. Specifically in the example of `concat`, the library function `(+)` is automatically reflected to the type level and is applied in the dependent type `Vec (n + m) a`.

The same verification for concatenating two lists can be specified using refinement type, as `val concat: #a:Type -> #n:nat -> #m:nat -> l1:Vec a -> l2:Vec a -> r:Vec a{len r = len l1 + len l2}`. Instead of indexing the length of the vector to form dependent types, we can specify the intuitive condition `len r = len l1 + len l2` as a logical predicate to the resulting type using subtyping.

4.1.1.2 Library and Documentation Supports

Dependent Haskell, extending Haskell with dependent types, has been oficially implemented in GHC 8.0 and is now a part of Haskell. As Haskell is a mature, industrial-strength programming language, Dependent Haskell inherits all the resources. Specifically, Haskell provides a central package archive called Hackage where packages are introduced, maintained, and supported. In addition, Hoogle is a holistic search engine for Haskell APIs, which provides advanced queries through function names as in `sum`, and even approximate type signatures, as in `int -> int -> int`. As Haskell becomes more popular in the real world, there emerges a plethora of online tutorials and textbooks with detailed explanations of syntax, concepts and algorithms. Since the introduction of Dependent Haskell in 2014, tutorials, such as schoolofhaskell, have gradually incorporated Dependent Haskell into their documentations. Official resources are also available on Richard Eisenberg's Github repository for his PHD dissertation (https://github.com/goldfirere/thesis).

However, as a completely redesigned language that is relatively new, F* is still in lack of many resources. Besides various conference publications, F* has an official website organizing all of its resources (https://www.fstar-lang.org). Most of the documentations are through its interactive tutorial (https://www.fstar-lang.org/tutorial/) and examples in the official FStar Github repository (https://github.com/FStarLang/FStar). In addition, source code for each library function can be found in the `ulib` folder of its Github repository (https://github.com/FStarLang/FStar/tree/master/ulib).

4.1.1.3 Proof Complexity

Regarding verification techniques, Dependent Haskell doesn't support any proof interaction or automation. Unlike most verifiers, it chooses to avoid compile-time termination checking but supports, through monads, most general programming effects like IO and Error etc. Verification in Dependent Haskell is guaranteed both sound and complete for total functions, but could get really tedious even for relatively simple sorting algorithms, like insertion sort, mergesort or quicksort.

(details on DH complexity on insertion sort, mergesort or quicksort are to be filled)

F*, on the other hand, provides a flexible combination of SMT-based proof automation and manually constructed proofs. In addition, F* can use SMT solver, such as Z3, to match SMT patterns provided in the manually provided lemmas to assist with user-constructed proofs. F* supports theorem proving through either enriching function types(officially referred as the intrinsic style) or by writing separate lemmas on properties (officially referred as the extrinsic style) [13]. It categorizes effects into `Tot`, `DIV` etc using Dijsktra monads, requires explicit annotation of the effect in type signatures and enforces termination checking on all total functions.

4.1.2 Approaches to Potentially Non-terminating Functions

To find out Dependent Haskell and F*'s different verification techniques towards non-terminating functions, we start from exploring the Peano division example (Complete implementations are provided in Appendix). Peano naturals are a simple way to encode natural numbers with the `Zero` base value and the `Succ` recursive incrementer. For example, natural number 0 is expressed as `Zero` and 1 is expressed as `Succ Zero` etc. Using the Peano division example, we want to prove the property that the quotient of a product of two natural numbers a, b and b is equal to a, i.e. $a * b/b = a$. As an educational example to simulate non-termination, we allow zero divisors in our definition of `division` and try to prove the property when the divisors are indeed positive, using the divergent division definition.

4.1.2.1 Dependent Haskell might be Complete towards Potentially Divergent Functions

Dependent Haskell doesn't require termination checking. With the help from type families and singletons, Dependent Haskell can successfully verify the Peano division property.

We first define the GADT `Peano` and the type level operations `+`, `-`, `*`, `/` in type families as follows:

```
data Peano where
  Zero :: Peano
  Succ :: Peano -> Peano

type family (a :: Peano) + (b :: Peano) :: Peano where ...
type family (m :: Peano) - (n :: Peano) :: Peano where ...
type family (a :: Peano) * (b :: Peano) :: Peano where ...
type family (a :: Peano) / (b :: Peano) :: Peano where ...
```

These four operations are used in the type signature of `peanoDivisionPf` defined below to prove the Peano division property.

Then we define the singleton type `SPeano` for natural numbers and specify the type signatures for the term level `plus`, `minus` and `mult` that are used in the actual implementation of the proof.

```
data SPeano n where
  SZero :: SPeano Zero
  SSucc :: SPeano n -> SPeano (Succ n)

plus :: SPeano m -> SPeano n -> SPeano (m + n)
minus :: SPeano m -> SPeano n -> SPeano (m - n)
mult :: SPeano m -> SPeano n -> SPeano (m * n)
```

Finally, we prove the property `(m * n)/n = m` for positive `n` in `peanoDivisionPf` using the three lemmas `plusZero`, `plusSucc` and `plusMinus`. We can define, using singletons, the positive divisor as `SPeano (Succ n)`, so the property becomes `(m * (Succ n)) / (Succ n) = m`. The `peanoDivisionPf` on the property and the three lemmas applied are shown below:

```
plusZero :: SPeano m -> m + Zero :~: m
plusZero SZero      = Refl
plusZero (SSucc m) =
   case (plusZero m) of Refl
-> Refl

plusSucc :: SPeano m -> SPeano n -> m + (Succ n) :~: Succ (m + n)
plusSucc SZero _     = Refl
plusSucc (SSucc m) n =
   case (plusSucc m n) of Refl
-> Refl

plusMinus :: SPeano m -> SPeano n -> (m + n) - n :~: m
plusMinus m SZero     =
     case (plusZero m) of Refl
  -> Refl
plusMinus m (SSucc n) =
   case (plusSucc m n) of Refl
-> case (plusMinus m n) of Refl
-> Refl

peanoDivisionPf :: SPeano m -> SPeano (Succ n) ->
  (m * (Succ n)) / (Succ n) :~: m
peanoDivisionPf SZero _                      = Refl
peanoDivisionPf (SSucc m) succ_n@(SSucc n) =
   case plusMinus (mult m succ_n) succ_n of Refl
-> case plusSucc (mult m succ_n) n of Refl
-> case peanoDivisionPf m succ_n of Refl
-> Refl
```

To understand the `peanoDivisionPf` proof, let's first discuss the three lemmas applied. The lemma `plusZero` proves that `Zero` is an additive identity of the Peano natural numbers. By definition of additive identity, the sum of every Peano natural number and the `Zero` Peano natural number is the Peano natural number itself, i.e. `m + Zero :~: m` for some Peano natural number `m`. The lemma `plusSucc` proves that `m + (Succ n) :~: Succ (m + n)`, or in algebraic form, `m + (1 + n) = 1 + (m + n)` for some Peano natural numbers `m` and `n`. Notice that `:~:` is a propositional GADT similar to the equal sign that denotes the logical equality in the type level. Finally, the lemma `plusMinus` proves the property on minus, where `(m + n) - n :~: m`.

Dependent Haskell provides the `@` syntax to reduce code verbosity. In `succ_n@(SSucc n)`, `(SSucc n)` is called an invisible argument, prefixing with `@`, and `succ_n` is the explicit value that stands for `SSucc n`. That is to say that every `(SSucc n)` in the following code can be expressed as `succ_n` [3]. Assume the inductive hypothesis that `(m * (Succ n)) / (Succ n) : : m`, the inductive step of the proof is shown step by step as follows:

```
  (Succ m * Succ n) / Succ n
  -- by definition of (*)
= (m * Succ n + Succ n) / Succ n
  -- by plusSucc
= Succ (m * Succ n + n) / Succ n
  -- by definition of (/)
= Succ Zero + (Succ (m * Succ n + n) - Succ n) / Succ n
  -- by definition of (+)
= Succ Zero + ((Succ (m * Succ n) + n) - Succ n) / Succ n
  -- by definition of (+)
= Succ (((Succ (m * Succ n) + n) - Succ n) / Succ n)
  -- by plusSucc (same as above, opposite direction)
= Succ (((m * Succ n) + Succ n - Succ n) / Succ n)
  -- by plusMinus
= Succ ((m * Succ n) / Succ n)
  -- by inductive hypothesis
= Succ m
```

In summary, Dependent Haskell doesn't require any level of termination checking at compile time, so divergent functions such as division by zero can still be reasoned easily and used to prove the Peano division property valid in logic. We still need more examples to determine whether Dependent Haskell is actually complete.

4.1.2.2 F* is Incomplete towards Potentially Divergent Functions

F*, based on the incomplete Hoare Logic, doesn't support extrinsic proofs on non-terminating functions. In Hoare Logic, the transition from one program state to another is encoded in Hoare triples. Hoare triple is of the form `P C Q`, where `P` and `Q` are called assertions in predicate logic denoting the precondition and postcondition, and `C` is the command that is being executed. Hoare logic is a mature way to categorize programming states, but unfortunately it is incomplete, as it cannot derive the logically valid specifications if the execution `C` cannot be proved to terminate. Instead, F* allows proving divergent programs intrinsically through type enrichment, as in the example of Peano Division.

First, we define the `peano` GADT as follows:

```
type peano: Type =
 | Zero : peano
 | Succ : peano -> peano
```

Then we show the type signatures of `plus`, `minus`, `mult` and `division` as in the ordinary operations on natural numbers as follows:

```
val maximal: int -> int -> Tot int
let maximal a b = if a > b then a else b

val plus: a:peano -> b:peano -> Tot (c:peano{toNat c >= toNat a && toNat c >=
    toNat b})
val minus: a:peano -> b:peano -> Tot (c:peano{toNat c = maximal (toNat a -
    toNat b) 0})
val mult: a:peano -> b:peano -> Tot (c:peano)
val division: a:peano -> b:peano -> Tot (c:peano)
```

Notice that `toNat` is a total function that takes in a `peano` GADT and returns the corresponding natural number, i.e. `toNat Zero = 0` and `toNat (Succ Zero) = 1` etc. Our verified `plus` function ensures that the sum is greater than or equal to both summands, and `minus` ensures that the difference is either `0` or equivalent to the difference of the two corresponding natural numbers.

For division, as we include the possibility that the divisor of `division` could be zero, we must mark the effect for `division` as `DIV`, meaning divergent. When we try to verify the property `(a*b)/b = a` following the same logic as in Dependent Haskell, we find ourselves in trouble, as `division` function with effect `DIV` cannot be directly reflected to verification conditions (VC) through F*'s monadic return typing rule. That is to say that we cannot use the divergent `division` function in `ensures`, i.e. the postcondition of the lemma.

In this educational Peano division example, we can easily modify the type signature to define a terminating `div_terminating` function. Thus, one workaround to prove the lemma `(a*b)/b = a` is by explicitly establishing the logical connection between the divergent function `division` and the total function `div_terminating` and then reason about the property using `div_terminating` in the postcondition of the lemma.

We show the functions `div_terminating` and `division` as well as the type signature for the proof `peanoDivPf` as follows:

```
val div_terminating: a:peano -> b:peano -> Pure (c:peano)
  (requires toNat b > 0)
  (ensures fun y -> True)
(decreases (toNat a))
let rec div_terminating a b =
  if a = Zero then Zero
  else let denom = (div_terminating (minus a b) b)
       in plus denom (Succ Zero)

val division: a:peano -> b:peano -> Div (c:peano)
  (requires True)
  (ensures (fun y -> toNat b > 0 ==> y == div_terminating a b))
let rec division a b =
  if a = Zero then Zero
  else let denom = (division (minus a b) b)
       in plus denom (Succ Zero)

val peanoDivPf: a:peano -> b:peano -> Lemma
  (requires (toNat b > 0))
  (ensures (div_terminating (mult a b) b == a))
```

Specifically, the total function `div_terminating` takes in two peano numbers dividend `a` and divisor `b` and returns the quotient `c`. Using the syntactic sugar `requires` and `ensures` as introduced in section 2.6.2.2, we specify the precondition to allow only nonzero divisors and the postcondition to be universally true for any inputs (`fun y -> True` is a lambda expression as in the familiar form `\y -> True` that is beyond the scope of the current discussion). To prove the totality of `div_terminating`, that is to prove that `div_terminating` indeed terminates, we specify our decreasing metric as `toNat a`. The metric `toNat a` is always decreasing since when we recursively call `div_terminating` on `(minus a b)` and `b`, we are in the `else` case where `a` is a nonzero natural number and by definition of `minus`, we always have `(minus a b) < a`. Therefore, with the decreasing metric, the function `div_terminating` is guaranteed to terminate by the well-ordering principle on natural numbers in algebra.

It follows that in the divergent function `division`, we can ensure that for any precondition, as long as the the divisor for `division` is nonzero, the result of `division` is equivalent to the result of `div_terminating`.

We call the proof `peanoDivPf` partial since the property only holds for a certain precondition. Also, though the introduction of the intermediary total function `div_terminating` completes the proof, this approach inevitably requires programmers to know the exact reason of non-termination. Therefore we believe that F* is not generally applicable, therefore incomplete, in verifying the potentially divergent functions in most practical programs.

4.2 Future Plans

4.2.1 Program Formalization

to be filled

4.2.1.1 Terms

to be filled

4.2.1.2 Values

to be filled

4.2.1.3 Evaluation Rules

to be filled

4.2.1.4 Small-Step Evaluator

to be filled

4.2.1.5 Big-Step Evaluator

to be filled

4.2.1.6 Typing Rules

to be filled

4.2.1.7 Progress and Preservation

to be filled

4.2.1.8 Simply Typed Lambda Calculus

to be filled

4.2.2 Approaches to Hard-to-prove Terminating Functions

We are in the progress of proving the equivalence of multi-step evaluator and big-step evaluator in simply typed lambda calculus. Simply typed lambda-calculus, or STLC, is a tiny core calculus incorporating the key concepts of functional abstraction, that is incorporated in most functional programming languages [12]. As STLC has enough structural complexity to illustrate interesting theoretical properties, it is commonly used to evaluate programming languages. STLC is based on a collection of base types, and for our purposes, we chose to study the simply typed lambda-calculus with booleans, which contains the following terms:

$$\begin{array}{rcl} term & ::= & x \\ & | & \lambda x : T.t \\ & | & t\,t \\ & | & True \\ & | & False \\ & | & if\ t_1\ then\ t_2\ else\ t_3 \end{array}$$

We also define values as the term where reduction stops, i.e. cannot evaluate/step further. Specifically in our case, we define the values as

$$\begin{array}{rcl} value & ::= & \lambda x : T.t \\ & | & True \\ & | & False \end{array}$$

Specifically, we first implemented the simple step function, following the operational semantics of the evaluation rules, that evaluate one step at a time. We then extend it to what's called a multi-step evaluation function that calls step function whenever the evaluated term is not a value. Finally, we implemented the big-step evaluation function that directly evaluates down to the final value. From type theory, we know that in simply typed lambda calculus, both functions evaluated down to values and are indeed equivalent. However, as there isn't a clear decreasing metric, we aren't sure if F* could be able to prove their totality. Hence, we are interested to explore if F*'s effect system is complete enough to handle this case. If so, we want to move on to examine what Haskell could adopt; otherwise if not, we will explore in detail the advantages and limitations of the current effect systems and conclude a recommendation for the future directions.

5 Related Work and Contributions

5.1 Language Comparison

There are many programming languages in use today and new ones are created every year. Various work has been done to compare programming languages. Through comparison, researchers find advantages and limitations in each and evaluate to propose insights for future research [4].

In the realm of programming languages, there are general-purposed programming languages like Pascal, Haskell and C/C++, interactive proof assistants like Coq, Agda and Isabelle/HOL, and SMT-based semi-automated program verification tools like Dafny, Vampire [6] and WhyML. Wiedijk compared a list of proof assistants on mathematical theorems, focusing on their strength of logic and level of automation [14] [15]. Feuer and Gehani edited papers comparing and assessing general purposed languages Ada, C and Pascal [4]. Filliâtre compared Vampire and Dafny in proving steps of the progress proof [11] [5].

(to be filled: summarize what they did and found)

As the needs grow for general purpose yet verification oriented programming languages, four functional programming languages have been recently invented, namely Dependent Haskell, Liquid Haskell, F* and Idris. As an ongoing field of research, to my best knowledge, these four communities generally conduct research separately and have rarely explored each other's approaches in-depth.

At the 45th ACM POPL Student Research Competition, Xu and Zhang presented their collaborated work on the comparison among three programing verification techniques in Dependent Haskell, Liquid Haskell and F* [16]. As an exploratory research, they found promising aspects in each language and proposed more detailed directions for further comparisons. In this thesis, we aim to provide an in-depth comparison between Dependent Haskell and F*, with a focus on their different verification approaches towards non-terminating functions. Through in-depth assessments of the verification techniques in both languages, we aim to find the more desirable approach and to recommend future directions for research in termination checking.

5.2 Dependent Haskell

to be filled

5.2.1 Functional Dependency

to be filled

5.2.2 Constrained Type Families

to be filled

5.2.3 Promoting Functions to Type Families in Haskell

to be filled

5.3 F*

to be filled

5.3.1 Relative Completeness in Hoare Logic

to be filled

5.3.2 Dijkstra Monads

to be filled

5.3.3 Hereditary Substitutions

to be filled

5.4 The Zombie Language

to be filled (Details about how the Zombie Language attempts to verify divergent functions)

Appendices

.1 Insersion Sort

.1.1 Dependent Haskell

to be filled

.1.2 F*

to be filled

.2 Peano Division

.2.1 Dependent Haskell

```haskell
{-# LANGUAGE GADTs, TypeInType, ScopedTypeVariables, StandaloneDeriving,
             TypeFamilies, TypeOperators, AllowAmbiguousTypes, TypeApplications,
             UndecidableInstances, DataKinds #-}

module PeanoDivisionDH where

import Data.Kind ( Type )
import Data.Type.Equality ( (:~:)(..) )
import Prelude hiding ( div )

data Peano where
  Zero :: Peano
  Succ :: Peano -> Peano
deriving (Eq, Ord, Show)

type family (a :: Peano) + (b :: Peano) :: Peano where
  Zero + b     = b
  (Succ a) + b = Succ (a + b)
infix 6 +

type family (m :: Peano) - (n :: Peano) :: Peano where
  Zero - _              = Zero
  m - Zero              = m
  (Succ m) - (Succ n)  = m - n
infix 6 -

type family (a :: Peano) * (b :: Peano) :: Peano where
  Zero * _     = Zero
  (Succ a) * b = a * b + b
infix 7 *

type family (a :: Peano) / (b :: Peano) :: Peano where
  Zero / _ = Zero
  m / n    = (Succ Zero) + (m - n) / n
infix 7 /

data SNat n where
  SZero :: SNat Zero
  SSucc :: SNat n -> SNat (Succ n)
deriving instance Show (SNat n)

plus :: SNat m -> SNat n -> SNat (m + n)
plus SZero n     = n
plus (SSucc m) n = SSucc (plus m n)

minus :: SNat m -> SNat n -> SNat (m - n)
minus SZero _              = SZero
```

```
minus m SZero             = m
minus (SSucc m) (SSucc n) = minus m n

mult :: SNat m -> SNat n -> SNat (m * n)
mult SZero _     = SZero
mult (SSucc m) n = plus (mult m n) n

plusZero :: SNat m -> m + Zero :~: m
plusZero SZero     = Refl
plusZero (SSucc m) = case (plusZero m) of Refl -> Refl

plusSucc :: SNat m -> SNat n -> m + (Succ n) :~: Succ (m + n)
plusSucc SZero _     = Refl
plusSucc (SSucc m) n = case (plusSucc m n) of Refl -> Refl

--   (m + (Succ n)) - (Succ n)
-- = Succ (m + n) - (Succ n), by plusSucc
-- = (m + n) - n, by definition of (-)
-- = m, by induction hypothesis
plusMinus :: SNat m -> SNat n -> (m + n) - n :~: m
plusMinus m SZero     = case (plusZero m) of Refl -> Refl
plusMinus m (SSucc n) =
   case (plusSucc m n) of Refl
-> case (plusMinus m n) of Refl
-> Refl

--   (Succ m * Succ n) / Succ n
-- = (m * Succ n + Succ n) / Succ n, by definition of (*)
-- = Succ (m * Succ n + n) / Succ n, by plusSucc
-- = Succ Zero + (Succ (m * Succ n + n) - Succ n) / Succ n, by definition of (/)
-- = Succ Zero + ((Succ (m * Succ n) + n) - Succ n) / Succ n, by definition of
    (+)
-- = Succ (((Succ (m * Succ n) + n) - Succ n) / Succ n), by definition of (+)
-- = Succ (((m * Succ n) + Succ n - Succ n) / Succ n), by plusSucc (same as
    above, opposite direction)
-- = Succ ((m * Succ n) / Succ n), by plusMinus
-- = Succ m, by induction hypothesis
peanoDivisionPf :: SNat m -> SNat (Succ n) -> (m * (Succ n)) / (Succ n) :~: m
peanoDivisionPf SZero _                   = Refl
peanoDivisionPf (SSucc m) succ_n@(SSucc n) =
   case plusMinus (mult m succ_n) succ_n of Refl
-> case plusSucc (mult m succ_n) n of Refl
-> case peanoDivisionPf m succ_n of Refl
-> Refl
```

.2.2 F*

```
module PeanoDivision

type peano: Type =
  | Zero : peano
```

```
  | Succ : peano -> peano

val toNat: peano -> Tot nat
let rec toNat a = match a with
  | Zero    -> 0
  | Succ a' -> 1 + toNat a'

val max: int -> int -> Tot int
let max a b = if a > b then a else b

val plus: a:peano -> b:peano ->
   Tot (c:peano{toNat c >= toNat a && toNat c >= toNat b})
let rec plus a b = match a with
  | Zero    -> b
  | Succ a' -> Succ (plus a' b)

val minus: a:peano -> b:peano ->
  Tot (c:peano{toNat c = max (toNat a - toNat b) 0})
let rec minus a b = match a with
  | Zero    -> Zero
  | Succ a' -> match b with
    | Zero    -> Succ a'
    | Succ b' -> minus a' b'

val mult: a:peano -> b:peano -> Tot (c:peano)
let rec mult a b = match a with
  | Zero    -> Zero
  | Succ a' -> plus (mult a' b) b

val div: a:peano -> b:peano{toNat b > 0} -> Tot (c:peano)
(decreases (toNat a))
let rec div a b =
  if a = Zero then Zero
  else plus (div (minus a b) b) (Succ Zero)

val plusZero: m:peano -> Lemma (ensures plus m Zero = m)
let rec plusZero m = match m with
  | Zero    -> ()
  | Succ m -> plusZero m

val plusSucc: m:peano -> n:peano -> Lemma (ensures plus m (Succ n) = Succ (plus
    m n))
let rec plusSucc m n = match m with
  | Zero    -> ()
  | Succ m -> plusSucc m n

val plusMinus: m:peano -> n:peano -> Lemma
(requires True)
(ensures minus (plus m n) n = m)
let rec plusMinus m n = match n with
  | Zero    -> plusZero m
```

```
  | Succ n -> match plusSucc m n with () -> plusMinus m n
  (*
    minus (plus a (Succ b')) (Succ b')
  = minus (Succ (plus a b')) (Succ b') by plusSucc a b'
  = minus (plus a b') b'
  = a by minusPlus a b'
  *)

val peanoDivPf: m:peano -> n:peano{toNat n > 0} -> Lemma
(requires True)
(ensures (div (mult m n) n == m))
let rec peanoDivPf m n = match m with
  | Zero   -> ()
  (*
    div (mult Zero b) b = div Zero b = Zero
  *)
  | Succ m ->
   match plusMinus (mult m n) n with ()
-> match peanoDivPf m n with
-> match plusSucc m Zero with
-> plusZero m
  (*
    div (mult (Succ a') b) b
  = div (plus (mult a' b) b) b
  = plus (div (minus (plus (mult a' b) b) b) b) (Succ Zero)
  = plus (div (mult a' b) b) (Succ Zero) by plusMinus (mult a' b) b
  = plus a' (Succ Zero) by peanoDivPf a' b
  = Succ (plus a' Zero) by plusSucc a' Zero
  = Succ a' by plusZero a'
  *)
```

References

[1] Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. Version 8.0.

[2] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[3] Richard A. Eisenberg. *Dependent Types in Haskell: Theory and Practice*. PhD thesis, University of Pennsylvania, 2016.

[4] Narain Gehani. *Comparing and assessing programming languages: Ada, C, and Pascal*. Prentice Hall, 1984.

[5] Sylvia Grewe, Sebastian Erdweg, and Mira Mezini. Automating proof steps of progress proofs: Comparing vampire and dafny. In *Vampire@ IJCAR*, pages 33–45, 2016.

[6] Laura Kovács and Andrei Voronkov. First-order theorem proving and vampire. In *International Conference on Computer Aided Verification*, pages 1–35. Springer, 2013.

[7] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, LPAR'16, pages 348–370. Springer Berlin Heidelberg, 2010.

[8] K Rustan M Leino. Automating induction with an smt solver. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 315–331. Springer, 2012.

[9] James McKinna. Why dependent types matter. In *ACM Sigplan Notices*, volume 41, pages 1–1. ACM, 2006.

[10] Ulf Norell. *Towards a practical programming language based on dependent type theory.* PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

[11] Benjamin C Pierce. *Types and programming languages.* MIT press, 2002.

[12] Benjamin C Pierce, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. Software foundations. *Webpage: http://www. cis. upenn. edu/bcpierce/sf/current/index. html*, 2010.

[13] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-Béguelin. Dependent types and multi-monadic effects in F*. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16. ACM, 2016.

[14] Freek Wiedijk. Comparing mathematical provers. In *International Conference on Mathematical Knowledge Management*, pages 188–202. Springer, 2003.

[15] Freek Wiedijk. *The seventeen provers of the world: Foreword by Dana S. Scott*, volume 3600. Springer, 2006.

[16] Rachel Xu and Xinyue Zhang. Comparisons among three programming verification techniques in dependent haskell, liquid haskell and f*. In *Proceedings of the 45th Annual ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL '18. ACM, 2018.